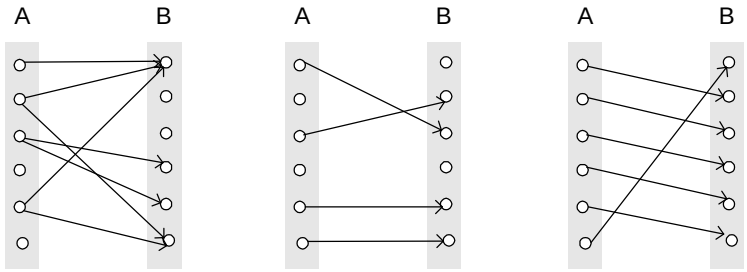


Das Relationenproblem

Version 1.0

Autor: Josef Hübl

Erstellt am: 25.11.2003

Geändert am: 30.06.2006

Von: Josef Hübl (Triple-S GmbH)

INHALTSVERZEICHNIS

Seite

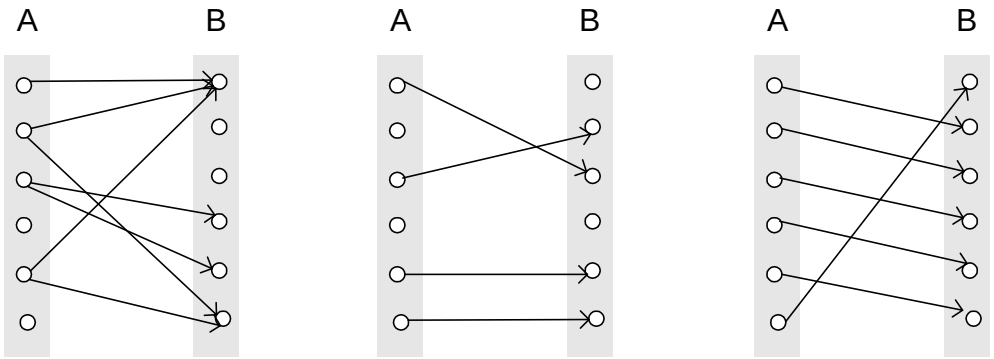
 Autor: Josef Hübl

1. Das Relationenproblem

1.1 Relationen bzw. Zuordnungen

Sind zwei Mengen A und B gegeben, dann heißt jede Teilmenge R von AxB eine Relation. D.h. einem Element aus A sind keine, genau ein oder mehrere Elemente aus B zugeordnet und umgekehrt. Eine Relation heißt injektiv, wenn einem Element aus A höchstens ein Element aus B zugeordnet ist und umgekehrt. Eine injektive Relation heißt eine Permutation, wenn jedem Element aus A genau ein Element aus B zugeordnet ist und Die Elemente von A und B geordnet sind.

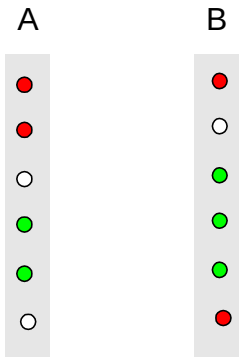
Relationen lassen sich besonders schön auch als gerichtete Graphen darstellen. Diese Graphen sind nicht nur gerichtet, sondern auch bipartit und zykelfrei. Umgekehrt kann über einen solchen Graphen auch eine Relation definiert werden, wenn jeder Knoten wenigstens einen Nachfolger oder Vorgänger und kein Knoten sowohl einen Nachfolger als auch einen Vorgänger besitzt.



Relation, injektive Relation und Permutation.

Werden den Ein-/Ausgängen eines Subzentrums Farben zugeordnet, so definieren diese eindeutig eine Relation zwischen den Ein-/Ausgängen, wenn einem Eingang alle gleichfarbigen Ausgänge zugeordnet werden.

 Autor: Josef Hübl



Eine über Farben definierte Relation

Allgemein definiert jeder Graph mit Kanten von den Ein- zu den Ausgängen genau eine Relation.

Da es Bausteine gibt die nur eine injektive Verschaltung zulassen muss auch bei den Relationen entsprechend unterschieden werden. Dazu werden die Elemente aus der Menge A als „multible“ gekennzeichnet, wenn der Baustein den entsprechenden Eingang beliebig verschalten kann. Für den Colored-Tree-Algorithmus bedeutet dies, dass in den als „multible“ gekennzeichneten Knoten Pfade zusammenlaufen können.

Achtung! Bausteine wir Benes32 und Benes16 können nur deshalb als elementare Bausteine modelliert werden, weil sie aufgrund zusätzlicher Schließer nicht nur Permutationen sondern beliebige injektive Relationen schalten können. Wären es dagegen reine Permutationsbausteine, so dürften sie nicht als elementare Bausteine verwendet werden, da der Routingalgorithmus als Ergebnis eine echte Teilpermutation liefern könnte, die der Baustein nicht verschalten kann. (Ein Nicht-Verschalten von Ausgängen gäbe es ja nicht!)

Die Farben der Ein-/Ausgänge einen Subzentrums definieren eine Relation zwischen diesen.

Das Relationenproblem:

Ist ein S-Netz gegeben, dessen äußere Ein-/Ausgänge teilweise gefärbt sind, so ist eine Erweiterung bzw. Fortsetzung der Färbung aller Ein-/Ausgänge gesucht, so dass das S-Netz formatiert ist und zu jedem gefärbten äußeren Ausgang mindestens ein Pfad von einem äußeren Eingang existiert, der identisch gefärbt ist

Das Relationenproblem kann in ein Colored-Tree-Problem für das transitive Komplement transformiert werden.

Die Kanten der Pfade die als Ergebnis des Colored-Tree-Problems gefunden werden, definieren eine Relation zwischen den Ein- und Ausgängen eines Subzentrums des originalen S-Netzes. Diese Relation ist injektiv, wenn kein Ausgang als „multible“ gekennzeichnet war. Ist die Relation nicht injektiv, so muss der zugehörige Baustein dann

 Autor: Josef Hübl

allerdings auch in der Lage sein, eine entsprechend allgemeine Relation zu verschalten. Beispiel: nxm-Matrix. Alternativ zur Kennzeichnung der Eingänge käme auch die Kennzeichnung eines Subzentrum als Ganzes (mit allen Eingängen) als „multible“ in Frage. Gemischte Subzentren müßten dann in vollständig „multible“ und nicht-multible aufgeteilt werden.)

1.2 Die Algorithmenkette

1. Verflachung des H-Netzes (über die Bausteinhierarchie)
2. Bildung der transitiven Hülle des flachen H-Netzes
3. Erzeugung des transitiven Komplements aus einer Kopie mit Zuordnung der Originalkoten.
4. Übertragen der Farben der äußeren Ein- und Ausgänge, wobei nicht benutzte äußere Ein-/Ausgänge mit einer zusätzlichen, nicht benutzten Farbe „eingefärbt“ werden.
5. Markiere alle Knoten, ausgenommen den Ein-/Ausgängen von Subzentren mit injekiven Relationen als „multible“.
6. Anwendung des erweiterten Colored-Tree-Algorithmus zum Ermitteln der Pfade.
7. Durchlaufen der gefunden Pfaden, dabei alle Kanten von Ein- zu Ausgängen eines Subzentrums extrahieren und dem originalen Subzentrum als Relation (Knotenpaar) zuordnen. (Hinweis: Die gefundenen Kanten existieren im originalen flachen H-Netz nicht!)

Der Umweg über das transitive Komplement ist auch deshalb notwendig, da allen fest vorgegebenen Kanten die im originalen S-Netz, einen Knoten gemeinsam haben, die selbe Farbe zugeordnet werden muss, damit mit dieses formatiert ist. Im transitiven Komplement erfolgt dies in einem Schritt durch die Zuweisung einer Farbe zum entsprechenden Subzentrum.

In den Algorithmus integriert ist eine Plausibilitätsprüfung der gefundenen Pfade, die zu einer harten Prüfung erweitert werden kann, wenn die Pfade von Endknoten die keine äußeren Eingänge sind, über andere Pfade, die diese Endknoten beinhalten bis zu äußeren Eingängen weiterverfolgt werden.

Als einfache Optimierung können bei der Erzeugung des transitiven Komplements alle neuen Subzentren weggelassen werden, die keine Ein- oder Ausgang besitzen. Ebenso können rekursiv alle Subzentren entfernt werden, die kein Vorgänger- oder kein nachfolgendes Subzentrum besitzen, da sie an keinen Pfad von den äußeren Aus- zu den äußeren

Eingängen beteiligt sein können. (Diese Reduzierung kann sowohl im originalen S-Netz als auch im transitiven Komplement angewandt werden.)

Ein weiteres Optimierungspotential ist dadurch gegeben, dass im resultierenden Graphen Ketten von Konten mit nur je einem Vorgänger und Nachfolger entfernt bzw. durch eine Kante überbrückt werden. Auf den gefundenen Pfaden sind solche neuen Kanten wieder durch die originale Knotenkette zu ersetzen, bevor mit der Ermittlung der Relationen fortgefahren wird.

Das Flag „multible“ muss verwendet werden, wenn für einen Baustein nur injektive Relationen zugelassen sind. Eine eventuell für Optimierungszwecke wichtige Erweiterung ist gegeben, wenn jedem Ein-/Ausgang gezielt ein „multible“-Flag zugeordnet werden kann. Dies kann eventuell genutzt werden um einen Algorithmus zu erstellen der eine Verflachung der Hierarchie zumindest teilweise überflüssig macht. D.h. die Größe der Netzwerke ist im Einzelfall wesentlich reduziert. Das „multible“-Flag kann auch zu einem Anzahl-Attribut erweitert werden.

 Autor: Josef Hübl

2. Implementierungsdetails

2.1 Hilfsroutinen der Klasse Graph

2.1.1 ExistingEdge()

Edge ExistingEdge(const Node &n1, const Node &n2) const;

ExistingEdge() überprüft, ob eine Kante bereits existiert und gibt sie zurück. Wenn sie nicht existiert, wird eine ungültige Kante (sie gehört zu keinem Graphen) zurückgeben.

2.1.2 CreateEdgeSingle()

Edge CreateEdgeSingle(const Node &s, const Node &t);

Edge CreateEdge(const Node &s, const Node &t);

Im Gegensatz zu CreateEdge() prüft die Funktion CreateEdgeSingle() vor dem Erzeugen einer neuen Kante erst, ob diese bereits existiert. CreateEdgeSingle() ist allerdings weniger performant als CreateEdge(), da im schlechtesten Fall alle Kanten von s überprüft werden müssen. CreateEdgeSingle() muss allerdings verwendet werden, wenn nicht gesichert ist, dass die neue Kante nicht existieren kann, da sonst ein Graph mit Mehrfachkanten entsteht.

2.1.3 AddTransitiveHull()

Edge AddTransitiveHull(const Node &n1, const Node &n2)

Edge AddTransitiveHull(const Node &n); // { return AddTransitiveHull(n, n); } ;

Edge AddTransitiveHull(const Edge &e); // { return AddTransitiveHull(e.source(), e.target());

AddTransitiveHull() bildet die transitive Hülle zwischen n1 und n2, d.h es werden alle noch fehlenden Kanten von den Vorgängern von n1 zu den Nachfolgern von n2 erzeugt. Dabei wird darauf geachtet, dass keine Mehrfachkanten erzeugt werden. Die erste neu erzeugte Kante wird zurückgegeben. Mit nextEdge() können alle weiteren neuen Kanten ermittelt werden.

2.1.4 DeleteTransitiveHull()

void DeleteTransitiveHull(const Node &n1, const Node &n2)

void DeleteTransitiveHull(const Node &n); // { DeleteTransitiveHull(n, n); } ;

void DeleteTransitiveHull(Edge &edge); // { DeleteTransitiveHull(e.source(), e.target()); };

DeleteTransitiveHull () löscht die transitive Hülle zwischen n1 und n2, d.h es werden alle Kanten von den Vorgängern von n1 zu den Nachfolgern von n2 gelöscht.

2.1.5 AddStar()

Node AddStar (const Node &n1, const Node &n2);

Node AddStar (const Edge &e);// { return AddStarBetween(e.source(), e.target()); };

AddStar () fügt einen neuen Knoten ein und verbindet ihn mit den Vorgängern von n1 und den Nachfolgern von n2. Der neue Knoten wird zurückgegeben.

2.1.6 AddStarBetween()

Node AddStarBetween(const Node &n);

AddStarBetween() fügt zwischen den Vorgängern v von n1 und n1 einen neuen Knoten v‘ ein und verbindet den Vorgänger v mit v‘ und v‘ mit n1 durch je eine Kante. Die Kante vom Vorgänger v zu n1 wird gelöscht. Analog wird mit allen Nachfolgern n von n2 verfahren.

Der erste neue Knoten wird zurückgegeben.

2.1.7 CreateNodeCopy()

Node CreateNodeCopy(const Node &n);

CreateNodeCopy() erzeugt einen neuen Knoten der mit allen Vorgängern von n bzw. Nachfolgern von n verbunden wird.

2.1.8 IsolateNode()

void IsolateNode(const Node &n);

IsolateNode() löscht alle Kanten, die von einem Konten ausgehen oder dort hin führen.

2.1.9 AddCopyOf ()

Node AddCopyOf(const Graph &g);

AddCopyOf() fügt einem Graphen eine Kopie der gegeben Graphen g hinzu. Der erste neu erzeugte Knoten wird zurück gegeben. AddCopyOf() wird auch verwendet um eine Kopie eines Graphen zu erstellen, wobei der Graph g einem leeren Graphen hinzugefügt wird.

2.1.10 Sonstiges

Die Routinen NextEdge(), NextIngoingEdge() und NextOutgoingEdge() sollten so überarbeitet werden, dass es möglich ist sie auch nach dem Löschen der aktuellen Kante aufzurufen, da sonst das Ausführen einer Aktion in einer Schleife über alle Kanten oder alle Vorgänger oder alle Nachfolger mit eventuellem Löschen der aktuellen Kanten, stets eine Ausnahmebehandlung erfordert.

2.2 Sonstiges

Objekthierarchien, H-Netze, Zuordnung der Subzentren zu den Objekten

Wichtig! Die Verflachung der Hierarchie muss in einer Art und Weise durchgeführt werden, so dass eine Zuordnung zu den elementaren Bausteinen möglich ist. D.h. die Verflachung muss so durchgeführt werden, dass sie dem rekursiven Durchlauf durch die Objekthierarchie entspricht. Am besten wird die Verflachung zusammen mit der Zuweisung der Subzentren zu den Objekten(Bausteinen) rekursiv durch die Bausteine selbst durchgeführt.

Da der „colored-tree"-Algorithmus im worst-case alle Pfadkombinationen durchprobiert und damit von der Ordnung O(p!) ist, liegt ein wesentliches Optinierungspotential darin den Graphen des S-Netzes in (Zusammenhangs-)Komponenten aufzuspalten und das Relationenproblem für jede Komponente getrennt zu lösen. Der Aufwand sinkt dabei auf O((p/k)!). Um den Zerfall in einzelne Komponenten zu forcieren können zuvor noch alle äußeren Ein-/und Ausgänge entsprechend der Anzahl ihrer aus- bzw. eingehenden Kanten vervielfacht (kopiert) werden, sofern die Kopien ihre Färbung beibehalten. Ist im Idealfall die Anzahl der innerhalb einer Komponente möglichen Pfade durch eine Konstante begrenzt, so führt das Aufteilen in Komponenten zu einem Algorithmus mit polynominalem Aufwand.

2.2.1 Die Printobjekte

Abgeleitet von ObjPrint und NodesNumbers

Zukünftig sollte jedem Konten eines Graphen ein Identifier hinzugefügt werden, da dies das debuggen wesentlich vereinfacht. Alternativ kann ein Compiler-Flag TRACE eingeführt werden, bei dessen Aktivierung mit dem jeweiligen Printobjekten gearbeitet wird.

2.2.2 Bausteine

Bausteine (Komponenten) sind elementare oder zusammengesetzte Objekte deren Struktur durch ein zugeordnetes H-Netz erfasst wird. Ein Baustein besitzt genau so viele Ein-/Ausgänge wie das zugeordnete H-Netz äußere Ein-/Ausgänge besitzt. Besteht ein Baustein aus Unterbausteinen, so ist diesem genau ein Subzentrum aus dem H-Netz des (Ober-) bausteins zugeordnet.

Jeder von der Klasse Baustein abgeleiteten Klasse ist ein Isomorphietyp zugeordent, der für alle Instanzen identisch ist. Da der Isomorphietyp zur Laufzeit ermittelt wird, wird er in einer statischen Variablen abgelegt die im *.cpp File der entsprechenden Klasse definiert ist. Die Ermittlung des Isomorphietypen erfolgt einmalig bei der Instanzierung des ersten Objekts der abgeleiteten Klasse.

Ein Unterbaustein kennt seinen umgebenden (Ober-)baustein nur, wenn er es von diesem mitgeteilt bekommt. Aus diesem Grunde wird im Konstruktor der this-Zeiger als „owner“ weitergegeben.

Zu beachten: Es wird davon ausgegangen, dass ein elementarer Baustein beliebige injektive Relationen schalten kann. Das bedeutet insbesondere, dass nicht unbedingt jedem Eingang auch ein Ausgang zugeordnet sein muss.

